Institut Galilée

INFO 2 :
FOURNIER Stéphane
ROUSSET Yohan

Compte rendu
du
Devoir de Graphe

Année 2009-2010

Table des matières

I. Introduction 5
II. Description du Code 6
A. Description du problème 6
B. Algorithmes 6
C. Architecture de l'application 8
1. La classe Sommet 8
2. La classe Arc 8
3. La classe Graphe 9
4. Les classes Fenetre et Panneau 10
D. Problèmes rencontrés 11
1. Problème au niveau de l'Applet 11
2. Problème d'indexage des sommets 11
III. Analyse de la Complexité 12
A. Implémentation par listes de sommets et d'arcs 12
1. Complexité en temps 12
2. Complexité en espace 13
B. Implémentation avec l'utilisation de la matrice d'adjacence. 13
1. Complexité en temps 13
2. Complexité en espace 14
C. Conclusion 14
IV. Mode d'emploi - Démonstration 15
A. Création d'un graphe 15
1. Directement par implémentation 15
2. Par l'interface Homme-Machine. 15
3. Par création d'un fichier d'instance 15
B. Manipulation des fichiers d'instances 15
1. Mise en forme 15
2. Bouton Chargement 16
3. Bouton Sauvegarder 16
C. Les boutons d'appel à Kosaraju-Sharir 17
1. Bouton par liste 17

2. Bouton par matrice 17
D. Exemple d'utilisation 17
E. Résultats obtenues sur les différentes instances 19
1. Instance exo2 20
2. Instance exo5B 21
3. Instance exo6 22
4. Instance exo7 23

I. Introduction

Dans le cadre de notre formation d'ingénieur, nous sommes amenés à suivre un cours d'algorithmique des graphes. A travers ce cours, nous abordons des notions inhérentes aux graphes. Nous sommes alors amenés à manipuler des graphes afin de déterminer, entres autres, le plus court chemin entre deux sommets, ou bien de déterminer les composantes fortement connexes.

Ainsi, notre application permet de déterminer ces composantes. Nous allons donc voir comment cela est fait en trois parties. Nous commencerons par décrire le code de notre application. Ensuite, nous déterminerons la complexité des différents algorithmes implémentés. Enfin, nous expliciterons l'utilisation de notre application à travers divers exemples.

II. Description du Code

Avant toute chose, il est important de préciser le problème que permet de résoudre notre application.

A. Description du problème

Le problème résolu par notre application est celui de la recherche des composantes fortement connexes d'un graphe orienté. Dans la suite, on utilisera la définition suivante des graphes : G = (X, U), où X est l'ensemble des sommets de G, et U l'ensemble de ses arcs.

Une composante fortement connexe est un ensemble de sommet où il existe, pour chaque sommet appartenant à une composante C, un chemin vers un autre sommet de C. Un exemple d'utilisation des composantes fortement connexes est, entres autres, celui de la détermination des sens interdit dans un quartier. En effet, il est important de prévoir à l'avance qu'on puisse atteindre, malgré les sens interdit, un point B à partir d'un point A.

B. Algorithmes

Un algorithme qui permet de déterminer les composantes fortement connexes d'un graphe est celui de Kosaraju-Sharir. Ce dernier se décompose en trois étapes :

- ✓ Exécution du DFS (Deep First Search : Recherche par Profondeur d'Abord) sur le graphe courant. La fonction renvoi l'ordre dans lequel les sommets ont été visités. La numération suffixe et préfixe est aussi effectuée.
- ✓ Inversion de l'orientation des arcs. On créé alors un graphe inversé G' = (X,U'), avec X l'ensemble des sommets de G et U' les arcs de G inversé.
- ✓ Exécution du DFS sur G' en partant successivement du sommet non visité de plus grand indice dans la numérotation suffixe.

L'algorithme définissant la fonction *DFS* ainsi que la fonction auxiliaire d'exploration *explore* appelé par *DFS* sont ceux du cours. Des fonctions supplémentaires ont été nécessaires pour l'implémentation, telle que la fonction déterminant le degré sortant de chaque sommet.

Algorithme du DFS

```
DFS (graphe G)
      k ← 0
      f ← 1
      Pour i = 0 à n Faire
            c(i) ← d+(i)
      Fin Pour
      Pour i = 0 à n Faire
            num(i) ← 0
      Fin Pour
      Pour i = 0 à n Faire
            Si num(i) = 0 Alors
                  k ← k +1
                  num(i) ← k
                  prefixe(i) ← k
                  explore(G, i)
            Fin Si
      Fin Pour
Fin DFS
```

Algorithme de explore

```
Procédure explore(graphe G, sommet i)
      Tant que c(i) > 0 Faire
            Sélectionner j le n(i)ème successeur de i
            n(i) ← c(i) - 1
            Si num(i) = 0 Alors
                  k ← k + 1
                  num(j) ← k
                  prefixe(j) ← k
                  explore(G, j)
            Fin Si
      Fin Tant que
      suffixe(i) ← f
      f ← f + 1
Fin explore
```

C. Architecture de l'application

L'application est composée de plusieurs classes qui vont être définies dans cette partie. Pour chacune d'elles, nous verrons les fonctions qu'elle comporte, ainsi qu'une brève explication pour chacune de ces fonctions. Pour de plus amples informations, il est recommandé de consulté les fichiers sources commentés fournis dans l'archives

1. La classe Sommet

La classe **Sommet** est la classe générique qui permet d'utiliser des sommets dans notre classe. Elle permet de mémoriser pour un sommet donné diverses informations importantes. Elle contient alors l'indice du sommet courant, deux entiers *x* et *y* qui sont les positions du sommet dans l'espace à deux dimensions, et, éventuellement, un objet de type quelconque (d'où l'utilisation de la généricité). Généralement, la classe est utilisée pour créer des sommets pouvant contenir des *String*. Cela sera utile lors du dessin du graphe réduit notamment.

class Sommet<T>
- T objetContenu; - int indice; - int x,y;
+ Sommet(int i) + Sommet(int i,int x,int y) + Sommet(int i, T o) + void setObjetContenu(T objetContenu) + int getX() + int getY() + void setX(int nx) + void setY(int ny) + boolean equals(Object obj) + String toString()

2. La classe Arc

La classe **Arc** utilise la classe **Sommet** précédemment définie. Un arc est alors composé de deux sommets représentant les sommets de départ et d'arrivée. Cette classe est utilisée par la classe **Graphe** dans le but de mémoriser les arcs d'un graphe donné.

class Arc <Sommet>
- Sommet depart - Sommet arrivee
+ Arc(Sommet depart, Sommet arrivee) + Sommet getDepart() + void setDepart(Sommet depart) + Sommet getArrivee() + void setArrivee(Sommet arrivee)

3. La classe Graphe

La classe **Graphe** est le cœur de notre application. C'est cette classe qui permet de créer des graphes et de les manipuler afin de calculer leur DFS, ou bien encore de déterminer leurs composantes fortement connexes grâce à l'algorithme de Kosaraju-Sharir.

C'est ainsi que la classe graphe comporte de nombreux attributs et fonctions permettant ces traitements.

En premier lieu, cette classe est composée de plusieurs constructeurs différents afin de permettre l'instanciation d'un graphe à partir de plusieurs structures de données différentes. Ainsi, on peut créer un graphe à partir d'un doublet « liste de sommets, liste d'arcs », ou bien alors à partir d'un triplet « matrice d'adjacence, nombre de sommets, nombre d'arcs ». Le constructeur qui utilise les listes créera également la matrice d'adjacence associée afin de pouvoir utiliser indifféremment l'algorithme de Kosaraju-Sharir sur les listes ou les matrices. En outre, lorsqu'un graphe est initialisé à partir d'une matrice, les arcs et sommets sont également créés afin de pouvoir utiliser l'algorithme de Kosaraju-Sharir utilisant les listes ou utilisant les matrices. Néanmoins, pour améliorer la complexité en espace, on peut simplement modifier le code de façon à n'utiliser que l'algorithme de Kosaraju-Sharir sur les listes si le graphe est initialisé avec des listes, et que l'algorithme de Kosaraju-Sharir sur les matrices, si le graphe est instancié à partir d'une matrice.

Dans un deuxième temps, une série d'accesseurs ont été implémentés afin de pouvoir éditer ou récupérer des valeurs importantes d'un graphe, tels que la liste des sommets, la liste des arcs, la matrice d'adjacence, un arc (i, j) particulier de la matrice, etc.

Ensuite, différents algorithmes sont implémentés afin de déterminer les composantes fortement connexes du graphe instancié. Ainsi, deux fonctions principales nommées *grapheReduit* et *grapheReduitMatrice* permettent de renvoyer le graphe des connexités fortes. Comme leur nom peut permettre de voir, *grapheReduit* s'appliquera en utilisant les listes de sommets et d'arcs, tandis que *grapheReduitMatrice* utilisera la matrice d'adjacence du graphe courant.

La fonction *grapheReduit* appelle dans un premier temps la fonction *Kosaraju-Sharir* afin de déterminer les composantes connexes. La fonction *Kosaraju-Sharir* appelle quant à elle appelle le DFS sur le graphe courant, elle inverse les arcs, puis appelle la fonction *DFSInverse().* Les fonctions d'exploration *explore* et *exploreInverse* sont implémentées afin d'être utilisées par *DFS()* et *DFSInverse().*

De la même manière, la fonction *grapheReduitMatrice* créer dans un premier temps la matrice d'adjacence du graphe courant. Ensuite, elle appelle *DFSMatrice* avant d'appeler *DFSMatriceInverse*. Ces deux fonctions permettent alors de déterminer les connexités fortes du graphe courant.

Dans les deux cas, les fonctions permettant de créer des graphes réduits parcourent la liste des arcs du graphe courant. Pour chaque arc analysé, elle regarde si les sommets de départs et d'arrivées ne sont pas dans la même composante fortement connexe. Si tel est le cas, alors il existe

un arc reliant deux composantes fortement connexes différentes. Cet arc est mémorisé et servira à l'affichage du graphe réduit. Lorsque le traitement est terminé, la fonction renvoi un graphe qui a pour sommets la liste des indices des composantes fortement connexes et contenant un objet *String* dans lesquelles sont mémorisés les sommets relatifs à la composante en question. Néanmoins, *grapheReduitMatrice* ne parcours pas directement une liste d'arcs. Elle doit en effet parcourir toutes la matrice à la recherche des arcs.

Les fonctions de sauvegarde et de chargement, ainsi que de traitement fichier seront traité plus tard (cf. IV.B)

4. Les classes Fenetre et Panneau

Implémentées dans les fichiers *Fenetre.java* et *Panneau.java*, ces classes permettent de créer une interface graphique afin de faciliter la manipulation de la classe Graphe. Ainsi, l'application s'utilise en lançant l'application comme suit : *java Fenetre*. Ceci aura pour effet de lancer l'interface graphique, et du même coup, l'application de manipulation des graphes.

Cette interface permet alors, par le biais de l'utilisation du menu déroulant dans la partie supérieur gauche, à l'utilisateur de :

- Ajouter un sommet au graphe courant par simple clique;
- Déplacer un sommet préalablement créer par glisser-déposer ;
- Ajouter un arc entre deux sommets par un clique sur le premier sommet (départ), et un second clique sur le second sommet (arrivée) ;
- Déplacer un graphe complet d'un point à un autre par glisser-déposer ;
- Effacer simplement le graphe courant.

Cette partie du devoir étant considérée comme bonus, nous n'en détaillerons pas d'avantage le code. Celui-ci est néanmoins accessible dans les fichiers implémentant les classes. Les algorithmes de tracé de graphes sont situés dans le fichier *Panneau.java*.

Néanmoins, un exemple d'utilisation de l'interface est disponible à la fin de ce rapport dans le mode d'emploi.

D. Problèmes rencontrés

Lors de l'implémentation de notre algorithme, nous avons rencontrés certains problèmes qui ont pu être résolu à temps.

1. Problème au niveau de l'Applet

Nous avons eu un problème lors de la création de l'Applet. En fait, lorsque nous avons commencé la création de l'écriture et de la lecture de fichier (d'instances notamment), nous avons rencontrés un problème de droit. Il était alors impossible d'ouvrir un fichier sur le disque de l'ordinateur. Ce problème de droit venait du fait qu'une Applet est censée, à la base, être une application de type « web ». Or, en java, une application de ce type doit être signée par son créateur. L'utilisateur client doit alors accepter le certificat crée par cette signature afin d'autorisé l'applet à enregistrer, ou lire, des fichiers sur le disque du client. Nous avons donc du convertir notre application afin de la faire fonctionner avec des JFrame. Le souci a alors été résolu.

2. Problème d'indexage des sommets

Typiquement, les algorithmes en pseudo-code comportent des tableaux indicés de *1* à *n.* Or, en programmation impérative, les tableaux sont indicés de *0* à *n-1*. Il a donc fallu faire très attention lors de l'implémentation des différents algorithmes à bien manipuler les indices. Dans notre architecture, les sommets ont des indices de *1* à n. Mais les tableaux de préfixes et de suffixes, eux sont indicés de *0* à *n-1.* Ainsi nous avons taché de faire attention à bien changer de base lorsque nous passons d'une structure à une autre. Et malgré toute notre volonté, il y a eu des erreurs faites lors de l'implémentation qu'il a fallu retrouver. Ceci a été fait et aujourd'hui l'application est fonctionnelle.

III. Analyse de la Complexité

Notre application permet d'utiliser deux types d'implémentation de l'algorithme de Kosaraju-Sharir. Nous verrons tout d'abord la complexité de la version qui utilise les listes de sommets et d'arcs. Enfin nous verrons l'implémentation par matrice d'adjacence.

A. Implémentation par listes de sommets et d'arcs

L'algorithme de Kosaraju-Sharir utilisant les listes d'arcs et de sommets est implémenté à travers la fonction *KosarajuSharir()*. Cette fonction permet applique les trois étapes vues précédemment (cf. II.B). Ainsi, nous allons analyser la complexité en temps et en espace de cet algorithme en analysant chacune de ses trois étapes.

1. Complexité en temps

Les trois étapes de l'algorithme étant appliquées à la suite, leurs complexités s'additionnent.

On notera *n* le nombre de sommet et *m* le nombre d'arc du graphe.

On a alors :

- *DFS()* est la fonction qui permet de remplir les tableaux d'entiers *suf, num,* et *pre.* En outre, elle initialise le tableau *c* des degrés sortant de chaque sommet, ce qui correspond à *m* tours de boucles. Dans DFS, chaque sommet est exploré au plus une fois (une fois qu'un sommet est exploré, il est en quelque sorte « marqué ». Il ne sera plus visité. En outre, pour récupérer le sommet à explorer, une fonction annexe est nécessaire : *cXiemeSuccesseur()*. Cette fonction est en O(m). Par ailleurs, chaque arc est exploré aussi au plus une fois (un arc reliant deux sommets, si le premier est marqué comme visité, l'arc ne pourra donc plus être emprunté). On alors une complexité pour DFS en $O(m^2n)$.
- Ensuite, la liste des arcs doit être inversée. Cela compte donc pour un parcours complet de cette liste pour en créer une nouvelle, inversé. Cela coûte *m* itérations.
- Enfin, la fonction *DFSInverse()* est appelée. Cette procédure possède une première boucle qui fait *n* tours de boucle. A chaque itération, elle récupère le successeur de plus grand suffixe. Ceci coute un parcours sur la liste des sommets, n tours de boucle. Or à chaque tour de boucle, l'algorithme verifie si le sommet appartient à la liste des sommets marqués. La fonction de recherche du sommet non marqués de plus suffixe est donc en $O(n^2)$. Si le sommet n'a pas été exploré (num[x] ==0), la fonction appelle la procédure *exploreInverse()* qui est le pendant de explore pour les arcs inversés. Or un sommet ne peut être exploré qu'une fois. Donc en pire cas, il peut y avoir *m* appels à explore (autant d'appels qu'il existe de successeurs, et donc d'arcs). On a alors une complexité pour le DFS sur les arcs inversé en $O(mn^3)$.

Pour résumer, la complexité de l'algorithme de Kosaraju-Sharir utilisant les listes d'éléments est de : $O(m^2n) + O(m) + O(mn^3)$. En outre, lorsqu'on ajoute la complexité de la création du graphe réduit (parcours complet de la liste des arcs, avec imbrication d'une boucle parcourant la liste des composantes fortement connexes (en pire cas, tous les sommets sont isolées, donc *n* tours de boucles) qui est en $O(mn)$, il vient une complexité totale de l'ordre de : $O(m^2n)+O(n)+ O(mn^3)$, soit $O(m^2n) + O(mn^3)$.

2. Complexité en espace

La complexité en espace est relativement simple à déterminer. En effet, si on suppose qu'on a *n* sommets et *m* arcs, alors on a une liste de sommets de *n* éléments. On a également une liste de *m* arcs. Sachant qu'un arc est composé de deux sommets, on a une complexité en taille de :

$$O(n) + O(m)$$

B. Implémentation avec l'utilisation de la matrice d'adjacence.

1. Complexité en temps

Pour créer le graphe réduit d'un graphe courant à partir de sa matrice d'adjacence, c'est la fonction *KosarajuSharirMatrice()* qui est appelée. Elle se décompose en deux appels de fonctions, à savoir *DFSMatrice()* et *DFSMatriceInverse()*. La première reprend dans les grandes lignes le même principe que *DFS()*. Elle appelle *remplirCMatrice()* qui est en $O(n^2)$ (parcours de la matrice complète pour déterminer les degrés sortant de chaque sommet). Puis, *InitialiseNum()* fait *n* tours de boucles à nouveau afin d'initialiser le tableau *num*. Ensuite, *DFSMatrice()* va faire *n* tours de boucles. En pire cas, elle fera *n* appels à *exploreMatrice()*. Cette fonction va alors appeler, tant que le sommet courant possède des successeurs, la fonction *cXiemeSuccesseurMatrice()* afin de déterminer le prochain sommet à explorer. Cette fonction fait alors très exactement *n* tours de boucles. Mais comme chaque sommet n'est exploré qu'une fois, il ne peut y avoir plus de *n* appels à explore. La complexité de *DFSMatrice()* est donc en $O(n^3)$.

Ensuite, contrairement à l'implémentation par listes, il n'est pas nécessaire d'inverser les arcs. En effet, il suffit de lire la matrice en intervertissant les lignes et les colonnes.

Enfin, *KosarajuSharirMatrice()* va appeler la fonction *DFSInverseMatrice()* afin de déterminer les composantes fortement connexes du graphe courant. De la même manière, cette fonction va initialiser le degré sortant de chacun des sommets. On est donc en $O(n^2)$. Ensuite, cette fonction va parcourir le graphe inversé en partant successivement du sommet de plus grand suffixe. Pour récupérer ce sommet, on a besoin de parcourir tous les sommets. On a alors *n* tours de boucles. Pour chaque sommet non visité, la fonction *exploreInverseMatrice()* va être appelée. En pire cas, elle peut être appelé *n* fois par le DFS inversé. Or cette fonction peut s'appeler récursivement si le sommet est non marqué. On est alors en $O(n^2)$. Donc la complexité de *DFSInverseMatrice ()* est en $O(n^3)$.

En pire cas, on est donc $O(n^3) + O(n^3)$, soit $O(n^3)$.

2. Complexité en espace

La complexité en espace de *Kosaraju-Sharir* correspond à la taille de la matrice d'adjacence. Cette taille ne dépend uniquement du nombre de sommets et est indépendante du nombres d'arcs. On a alors une complexité en taille de :

$$O(n^2)$$

C. Conclusion

Il existe plusieurs façons d'implémenter l'algorithme de Kosaraju-Sharir. Selon les structures de données choisies, les complexités en temps et en espace sont affectées immédiatement. Ainsi, l'implémentation par matrice d'adjacence en n^2 est lourde. Mais si elle peut faire gagner du temps lors de l'exécution de l'algorithme, elle peut être très utile.

Il faut alors choisir la structure de données la plus adaptés à l'utilisation que l'on souhaite faire de l'algorithme. Ainsi, si le nombre d'arcs est relativement faible par rapport au nombre de sommets, il sera préférable d'utiliser l'implémentation par listes. En revanche, si le nombre d'arcs est relativement élevé par rapport au nombre de sommets, malgré le coût en espace, il sera préférable d'utiliser l'implémentation par matrice dont la complexité est indépendante du nombre d'arcs.

IV. Mode d'emploi - Démonstration

Pas à pas, nous allons expliquer comment utiliser notre application. Commençons par la création d'un graphe.

A. Création d'un graphe

1. Directement par implémentation

On peut créer un graphe en utilisant les matrices d'adjacences ou les listes de sommets et d'arcs directement en le codant. Ainsi, des exemples de graphes ont été implémentés dans le fichier *Fenetre.java* (à partir de la ligne 425). Seul l'exercice 5 du TD3 est décommenté et est donc créé à l'instanciation de la fenêtre. Mais on peut alors simplement modifier le programme afin d'utiliser les graphes implémentés. L'utilisateur peut même s'il le désire créer le graphe de son choix en créant des sommets qu'il ajoutera à une *ArrayList* de Sommet, et des arcs qu'il ajoutera à une *ArrayList* d'Arc. Une fois cela fait, il ne lui reste plus qu'a construire le graphe de son choix. L'utilisateur prendra le soin nécessaire à indicer ses sommets de 1 à n pour n sommets. Les indices des sommets doivent être deux à deux différents et doivent être indicés de façon continue (pas de 1 – 2 – 4, etc.).

L'utilisateur peut aussi directement créer la matrice d'adjacence. Il suffira d'appeler le constructeur de Graphe adéquat en n'omettant pas de fournir le nombre de sommets et le nombre d'arcs contenus dans sa matrice.

2. Par l'interface Homme-Machine.

Afin que l'utilisateur puisse facilement créer un graphe sans avoir recourt au changement du code source, nous avons mis en place un système interactif à travers l'utilisation de l'interface graphique. Ainsi l'utilisateur pourra créer un graphe directement a partir de l'interface. Pour cela il devra d'abord effacer le graphe existant en se positionnant sur « Effacer ». Une fois la fenêtre vidée, il devra se positionner dans l'onglet « ajouter sommet » du menu déroulant. Un sommet est alors créé et positionné à l'endroit ou l'utilisateur relâche la souris. Une fois le minimum de deux sommets créés, il pourra alors créer des arcs en se mettant dans l'onglet « ajouter arc » du menu déroulant. Selon le même principe il cliquera d'abord sur le sommet de départ de l'arc puis il sélectionnera le sommet d'arrivé. Pour cela le programme utilise une fonction *« selectionne »* qui permet de trouver le sommet le plus proche de là ou a cliqué l'utilisateur.

3. Par création d'un fichier d'instance

Un graphe peut-être créé à partir d'un fichier texte. Celui-ci doit avoir une forme spéciale afin que le programme puisse créer le graphe que l'utilisateur souhaite. C'est ce que nous allons voir dans la partie suivante.

B. Manipulation des fichiers d'instances

1. Mise en forme

Le fichier d'instance est formaté comme suis :

```
Nombre de sommet : n
Nombre d'arc : m
```

```
arc( numéro du sommet de départ , numéro du sommet d'arrive )
Matrice d'adjacence:
```

L'utilisateur mettra donc dans un fichier les informations ci-dessus. Il pourra alors charger le fichier par le biais de l'interface graphique. L'utilisateur pourra aussi choisir entre l'utilisation du type « `arc` » dans le fichier ou alors le choix de construction d'un graphique par la « `matrice d'adjacence` ».

Exemple :

```
Nombre de sommet : 8
Nombre d'arc : 10
arc( 1 , 2 )
arc( 1 , 3 )
arc( 2 , 4 )
arc( 2 , 5 )
arc( 4 , 3 )
arc( 4 , 5 )
arc( 5 , 6 )
arc( 5 , 7 )
arc( 7 , 6 )
arc( 8 , 7 )
Matrice d'adjacence:
0 1 1 0 0 0 0 0
0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0
0 0 1 0 1 0 0 0
0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
```

2. Bouton Chargement

Le bouton chargement de l'interface graphique permet de récupérer le contenu du champ texte situé à coté de celui-ci. On appel alors la fonction « chargement » de la class Graphe. Cette fonction permet de récupérer le contenu du fichier dans une chaine de caractères. Celle-ci est utilisée comme argument de la fonction « traitementFichier ».

La fonction « traitementFichier » utilise un *String Tokenizer*. C'est-à-dire que le fichier sera analysé par mots. On pourra alors récupérer le nombre de sommets, et le nombre d'arcs afin d'initialiser les listes aux bonnes valeurs. La fonction peut alors adopter deux comportements différents. Soit le fichier contient la liste d'arcs du graphe. Les arcs sont alors récupérés et reconstruits. Si la liste d'arcs n'est pas contenue dans le fichier, un second comportement est adopté. Celui-ci permet alors de créer les arcs du graphe à partir de la matrice d'adjacence.

3. Bouton Sauvegarder

Le bouton sauvegarde de l'interface graphique permet de récupérer le contenu du champ texte situé à coté de celui-ci. On appel alors la fonction « sauvegarder » de la class Graphe avec

comme argument la chaine de caractères du champ texte. Le programme va alors crée un fichier du nom de la chaine de caractères. Puis le programme qui contient la trame du fichier d'instance va alors compléter les éléments vides grâce aux listes de sommets et d'arcs du graphe. La matrice d'adjacence sera créée grâce à un appel à la fonction « creeMatrice » de la classe Graphe.

C. Les boutons d'appel à Kosaraju-Sharir

1. Bouton par liste

Une fois se bouton appelé on va appeler la fonction « grapheReduit » de la classe Graphe. Comme vu dans la classe Graphe ci-dessus. La fonction « *grapheReduit »* appelle dans un premier temps la fonction « *KosarajuSharir »* afin de déterminer les composantes connexes. La fonction « *Kosaraju-Sharir »* appelle quant à elle appelle le DFS sur le graphe courant, elle inverse les arcs, puis appelle la fonction « *DFSInverse ».* Les fonctions d'exploration *explore* et « *exploreInverse »* sont implémentées afin d'être utilisées par *DFS()* et « *DFSInverse ».* On assigne alors le nouveau graphe réduit au graphe courant en ajoutant la liste des composantes fortement connexes aux noms des nouveaux sommets.

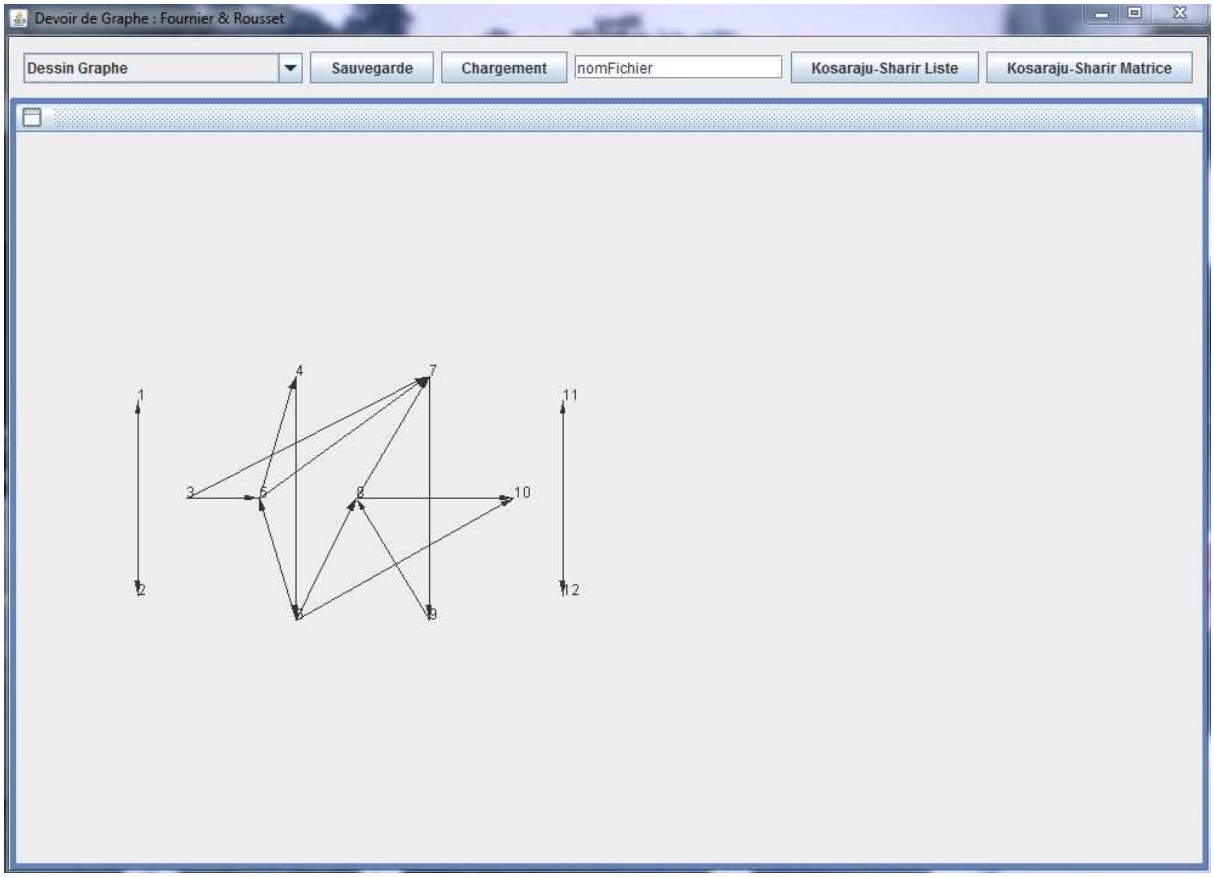
2. Bouton par matrice

Une fois se bouton appelé on va appeler la fonction « grapheReduitMatrice » de la classe Graphe. Comme vu dans la classe Graphe ci-dessus. La fonction « grapheReduitMatrice» appelle dans un premier temps la fonction « *KosarajuSharirMatrice »* afin de déterminer les composantes connexes. La fonction « *KosarajuSharirMatrice »* appelle quant à elle appelle la fonction « DFSMatrice » sur le graphe courant, puis appelle la fonction « *DFSInverseMatrice ».* Les fonctions d'exploration « *exploreMatrice »* et « *exploreInverseMatrice »* sont implémentées afin d'être utilisées par *DFSMatrice()* et « *DFSInverseMatrice ».* On assigne alors le nouveau graphe réduit au graphe courant en ajoutant la liste des composantes fortement connexes aux noms des nouveaux sommets.

D. Exemple d'utilisation

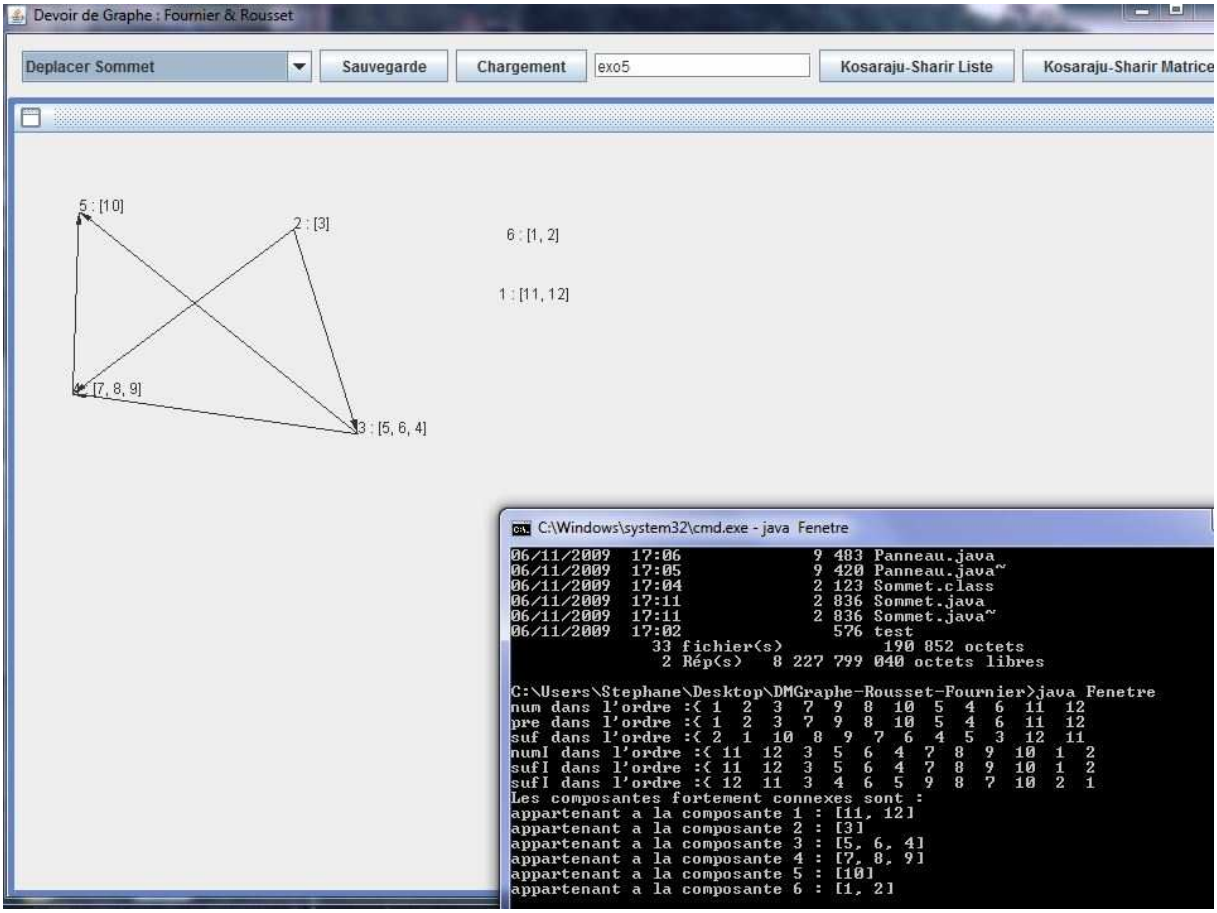
Afin d'illustrer nos dires nous joignions un exemple d'utilisation de notre programme.

Nous afin donc choisit d'initialiser un Graphe semblable a celui vu dans le TD3 exercice n°5 (Figure1).



Graphe du TD3 exercice n°5 (Figure 1)

Une fois le Graphe initialisé grâce à l'un des differents modes (fichier d'instance, implémentation, ou par l'IHM). Nous cliquons donc sur le bouton Kosaraju-Sharir Liste. Une fois la fonction executée nous obtenons le resultat ci-dessous (Figure2).


Devoir de Graphe : Fournier & Rousset
Deplacer Sommet
Sauvegarde
Chargement
exo5
Kosaraju-Sharir Liste
Kosaraju-Sharir Matrice
5 : [10]
2 : [3]
6 : [1, 2]
1 : [11, 12]
[7, 8, 9]
3 : [5, 6, 4]
C:\Windows\system32\cmd.exe - java Fenetre
06/11/2009 17:06 9 483 Panneau.java
06/11/2009 17:05 9 420 Panneau.java~
06/11/2009 17:04 2 123 Sommet.class
06/11/2009 17:11 2 836 Sommet.java
06/11/2009 17:11 2 836 Sommet.java~
06/11/2009 17:02 576 test
33 fichier(s) 190 852 octets
2 Rép(s) 8 227 799 040 octets libres
C:\Users\Stephane\Desktop\DMGraphe-Rousset-Fournier>java Fenetre
num dans l'ordre :< 1 2 3 7 9 8 10 5 4 6 11 12
pre dans l'ordre :< 1 2 3 7 9 8 10 5 4 6 11 12
suf dans l'ordre :< 2 1 10 8 9 7 6 4 5 3 12 11
numI dans l'ordre :< 11 12 3 5 6 4 7 8 9 10 1 2
sufI dans l'ordre :< 11 12 3 5 6 4 7 8 9 10 1 2
sufI dans l'ordre :< 12 11 3 4 6 5 9 8 7 10 2 1
Les composantes fortement connexes sont :
appartenant a la composante 1 : [11, 12]
appartenant a la composante 2 : [3]
appartenant a la composante 3 : [5, 6, 4]
appartenant a la composante 4 : [7, 8, 9]
appartenant a la composante 5 : [10]
appartenant a la composante 6 : [1, 2]

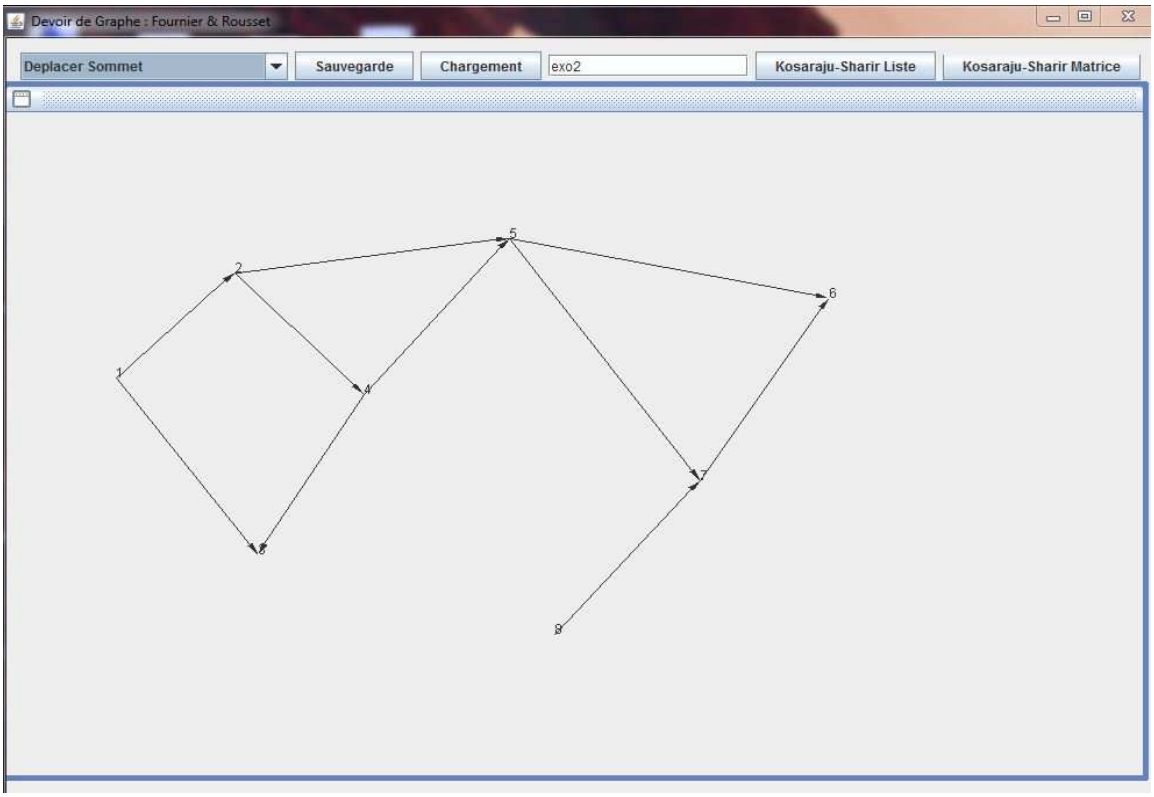
Graphe réduit du Graphe Figure1 (Figure2)

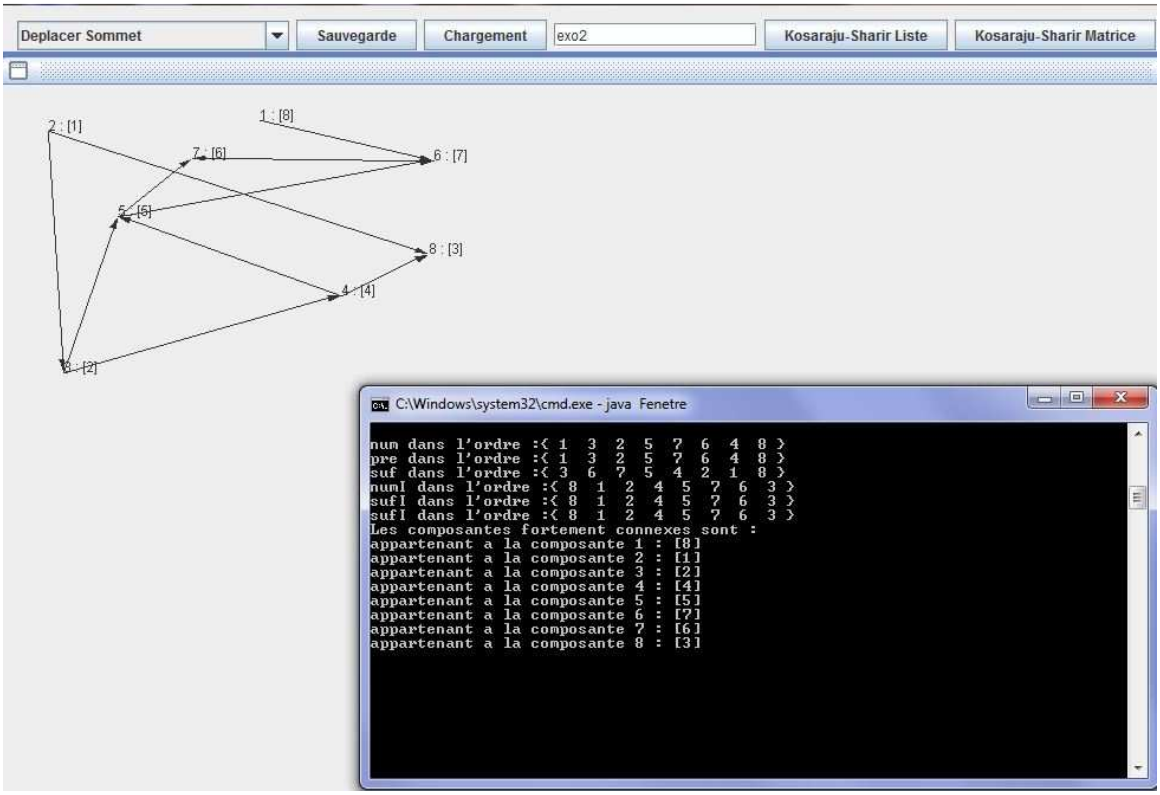
Nous voyons que nous obtenons bien le même graphe réduit que lors du TD. Nous avons ainsi pu valider notre programme grâce a plusieur exemple de reduction de Graphe comme celui-ci. Nous avons aussi laisser les différents affichage quand nous effectuons le DFS et lorsque nous trouvons les composantes connexes. L'utilisateur pourra alors refaire lui-même la réduction du Graphe à la main et à l'identique. A noter que l'utilisation de Kosaraju-Sharir Matrice donne le meme resultat que Figure2.

E. Résultats obtenues sur les différentes instances

Les instances que nous présentons sont des graphes proposés dans l'énoncé du TD3. Ils sont reconnaissables par numéro d'exercice.

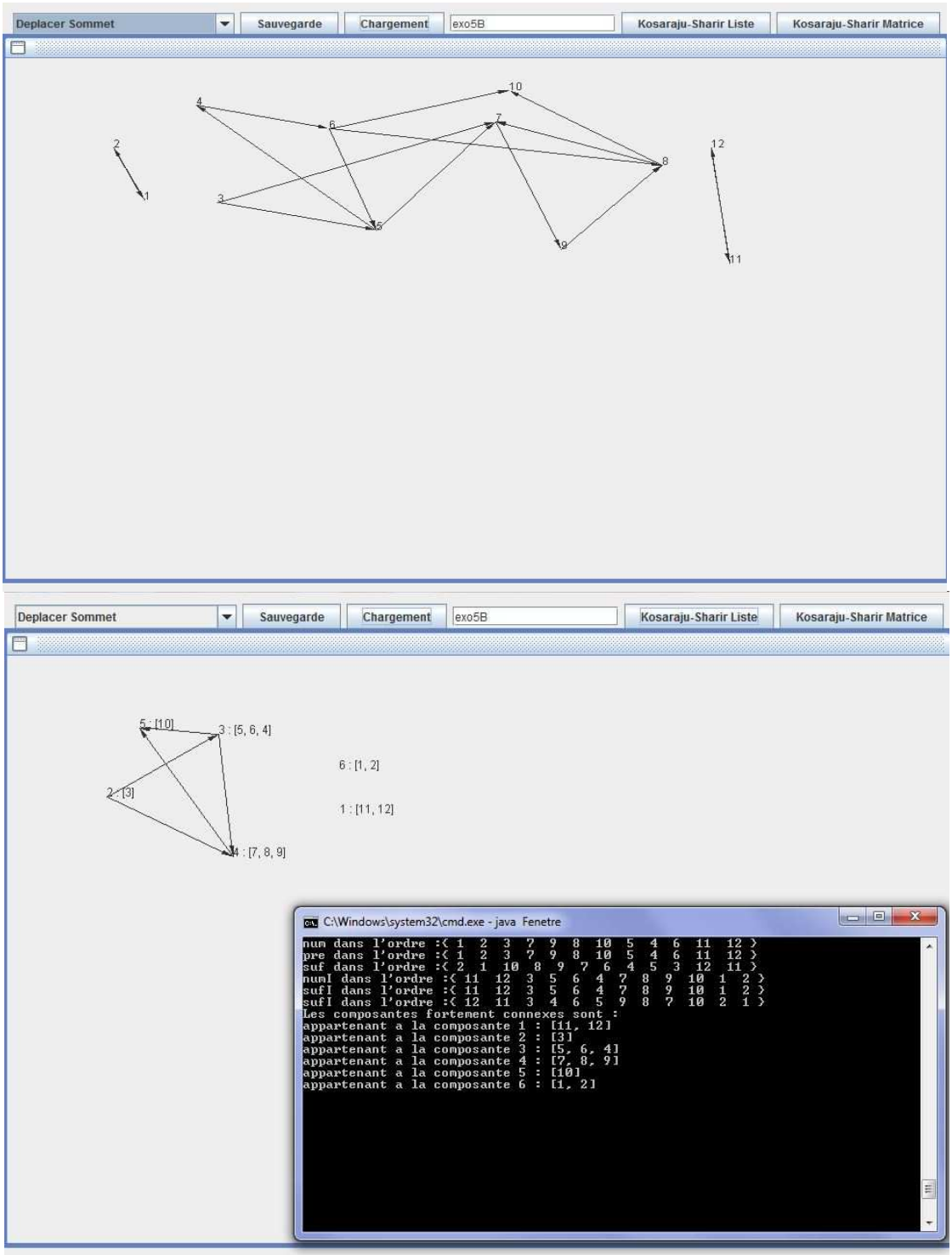
1. Instance exo2





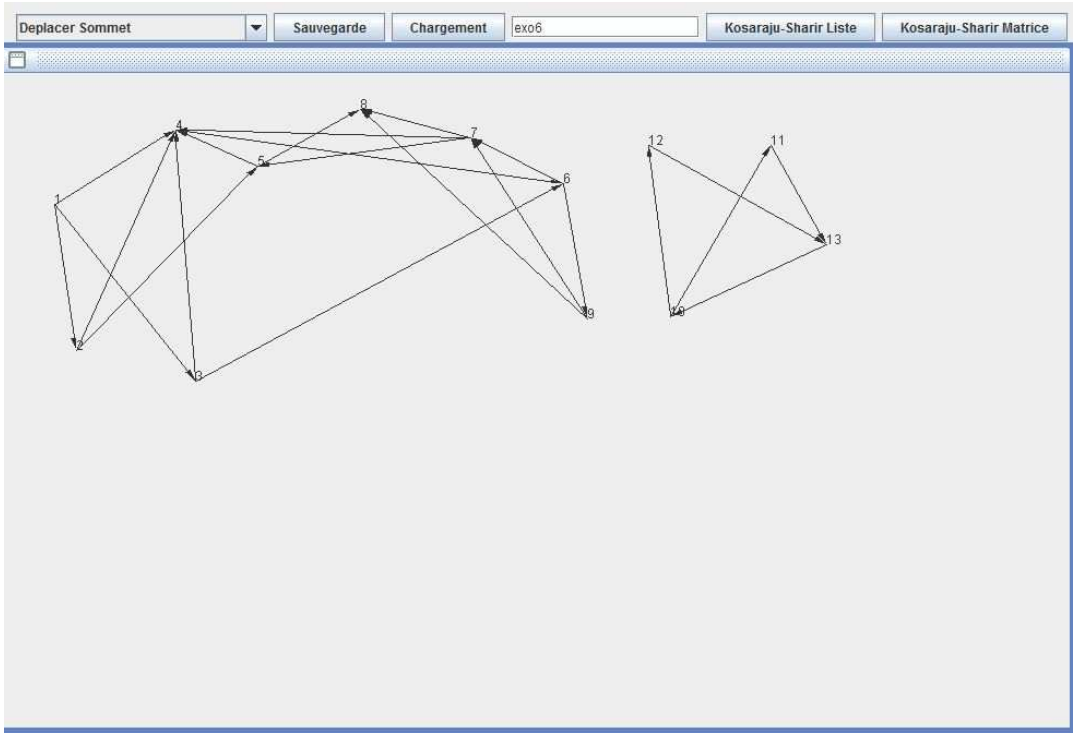
2. Instance exo5B

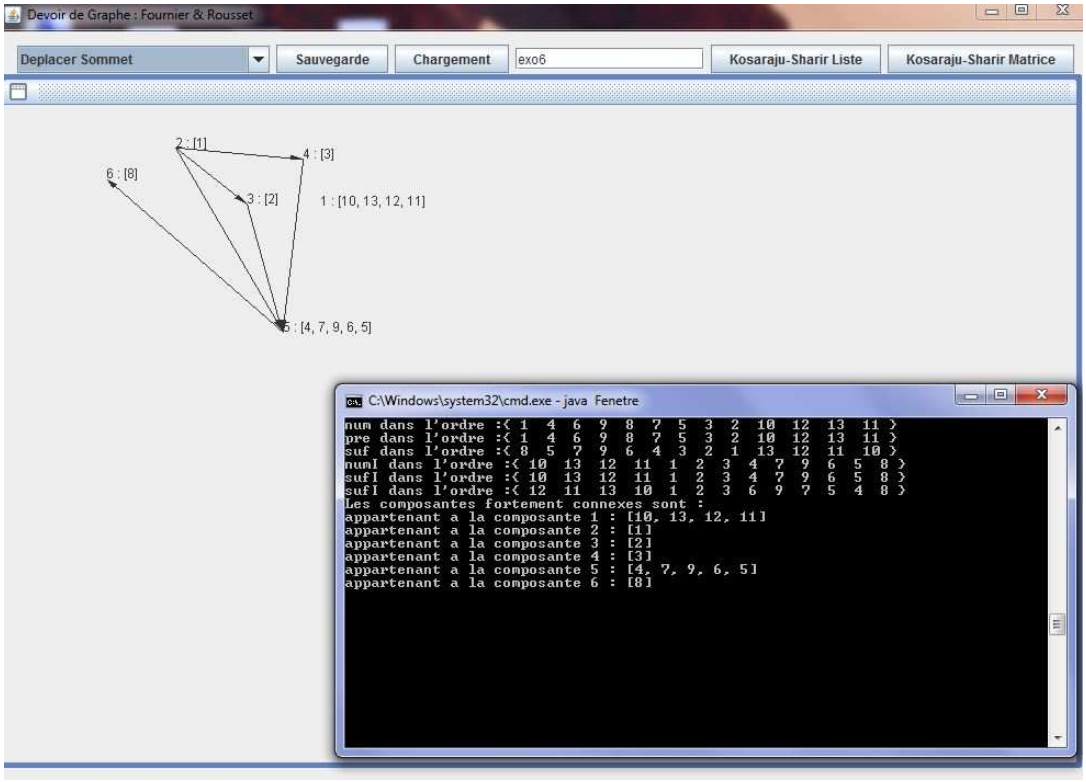
Cette instance est différente de l'instance exo5 utilisée dans la partie du mode d'emploi. En effet, cette instance n'utilise que le nombre de sommets, d'arcs, et la matrice d'adjacence pour être créée. Voici le résultat, il est le même qu'avec l'utilisation de exo5 :



3. Instance exo6

Cette instance utilise le bouton Kosaraju-Sharir Matrice.





4. Instance exo7